

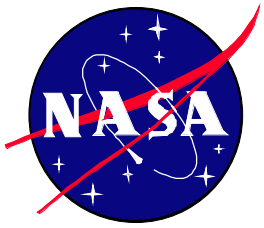


# Deorbit Flight Software Demonstration Lessons Learned

## ENGINEERING DIRECTORATE

## AEROSCIENCE AND FLIGHT MECHANICS DIVISION

**19 January 1998**




**National Aeronautics and Space Administration**

**Lyndon B. Johnson Space Center**
**Houston, TX**



# Deorbit Flight Software Demonstration Lessons Learned

**Prepared By:**

**Denise M. DiFilippo**
**G. B. Tech, Incorporated**

**Approved By:**

**David A. Petri**
**GN&C Rapid Development Lab Manager**
**Aeroscience and Flight Mechanics Division**
**NASA/Johnson Space Center**

**James P. Ledet**
**Code Q RTOP Project Manager**
**Aeroscience and Flight Mechanics Division**
**NASA/Johnson Space Center**

**Concurred By:**

**Aldo J. Bordano, Chief**
**Aeroscience and Flight Mechanics Division**
**NASA/Johnson Space Center**

**Karen D. Frank, Chief**
**GN&C Development & Test Branch**
**Aeroscience and Flight Mechanics Division**
**NASA/Johnson Space Center**

This Page Intentionally Blank

Table of Contents

## Acronyms and Abbreviations

| | |
|---|---|
| AFMD | Aeroscience and Flight Mechanics Division |
| API | Application Programming Interface |
| COTS | commercial, off-the-shelf |
| EGI | Embedded GPS/INS |
| FCOS | Flight Control Operating System |
| FSSR | Shuttle Flight Subsystem Software Requirements |
| FY | Fiscal Year |
| GN&C | Guidance, Navigation & Control |
| GPS/INS | Global Positioning System/Inertial Navigation System |
| HIL | hardware-in-the-loop |
| ISO | International Standards Organization |
| IV&V | Independent Verification and Validation |
| JSC | Johnson Space Center |
| NASA | National Aeronautics and Space Administration |
| OMS | Orbital Maneuvering System |
| RDL | Rapid Development Laboratory |
| SLOC | Source Lines of Code |
| SR&QA | Software Reliability and Quality Assurance |

## 1.0 Introduction

The National Aeronautics and Space Administration (NASA) is studying the feasibility of making major upgrades to the Space Shuttle, including the Shuttle avionics. A decision to upgrade the avionics hardware could carry with it the need to upgrade the on-board flight software and the associated ground based test, training and support software and hardware systems.

What is the best overall approach if such a software system upgrade is required? One possibility would be to take advantage of this opportunity to rewrite the flight software using modern tools, technology and techniques. It is difficult to accurately evaluate the potential impact of such a large and unique effort, in costs, schedules and risks, although it certainly would be a very large and complex endeavor.

The Aeroscience and Flight Mechanics Division (AFMD) at NASA's Johnson Space Center (JSC) Engineering Directorate is exploring ways of producing Guidance, Navigation & Control (GN&C) systems more efficiently and cost effectively. A significant portion of this effort is software development, integration, testing and verification.

A natural synergy thus exists, in which AFMD could apply their evolving facilities and techniques while providing empirical data to the analysis of effort and risk involved in upgrading the Orbiter flight software.

AFMD has established the GN&C Rapid Development Laboratory (RDL), a hardware/software facility designed to take a GN&C design project from initial inception through hardware-in-the-loop (HIL) testing and perform final GN&C system verification. The operations approach for the RDL emphasizes the use of commercial, off-the-shelf (COTS) software products to implement the GN&C algorithms in the form of graphical data flow diagrams, to automatically generate source code from these diagrams and to execute the software in a real-time, HIL environment, following a Rapid Development methodology.

The Flight Software Demonstration Project constructed a core team of systems, software, and GN&C domain experts. The team identified a significant yet manageable subset of the Space Shuttle GN&C on-board flight software, the deorbit flight phase, to implement and demonstrate. Making use of the facilities and philosophy of the RDL, the GN&C application software was designed, implemented, and tested. Multiple software engineering environments were explored and compared. Areas where the RDL processes and infrastructure needed augmentation or improvement were identified, and these evolved along with the flight software itself, throughout the project. The resulting flight software was released for Independent Verification and Validation (IV&V), and has been widely demonstrated to NASA management. The final demonstration included pilot-in-the-loop capability and OMS actuator in the loop, with the flight software executing on a target flight computer.

After observing and participating in the test project for slightly more than one year, the team has learned some lessons which will help us improve and refine the methodology. These lessons are described in this report.

## 2.0 The Test Project

The GN&C Orbiter Upgrades Deorbit Phase Flight Software Demonstration Project was identified and chosen as our test project. This project was planned to investigate the ability of the Aeroscience and Flight Mechanics Division, and the Rapid Development Laboratory, to quickly and effectively design, implement, test and deliver high quality GN&C flight software. In addition, it expands the knowledge base of the RDL by implementing a flight phase not previously developed in the RDL. This project was an appropriate test case (for the Rapid Development Methodology) for several reasons:

- It was just starting up, which allowed the rapid development research team to participate from the beginning.
- The project team was interested in the methodology and willing to use it.
- The project was appropriately sized to challenge the methodology while giving quick results.
- Using the methodology for this project is consistent with the goal of the RDL to explore ways of producing GN&C systems more efficiently and effectively.

The project was not, however, an ideal choice as a test case because:

- Much of the requirements discovery phase was irrelevant, since system requirements were based on existing FSSRs
- The project has high visibility and a demanding schedule. As such, it provided minimal time for reflection, rework, and comparison implementation. The project management did address this to the extent possible.

The core project team was constructed in August 1996. The first demonstration version was delivered in December 1996. A completely functional deorbit phase system was delivered for IV&V in September 1997.

## 3.0 What We Learned

This project began with high expectations, both for delivering flight software and for learning. The lab, team and division positioned this task as a project to gain understanding about RDL capabilities, Rapid Development methodology, tools for flight software development, and costs and feasibility of a major upgrade of the Orbiter GN&C flight systems. It has been a high energy work environment. The types of questions we asked tend to be moving targets, for both questions and answers. But there is no doubt that the lessons learned were substantial. This section highlights and describes some of the most significant of these.

## 3.1 Project Successes

Lessons learned reports tend to focus critically on projects. We turn a critical eye on each project in order to learn how to improve ourselves, our lab, and our processes and position the next project for success. Some of our lessons are about things that were done very well and should be emulated in later projects. Since the successes tend to be quickly absorbed into the local process culture, and deficiencies cry out to be corrected so that they do not become standard procedure, the nature of this kind of report is often to call attention to the deficiencies observed.

Before beginning the critical look, however, it is important to recognize the successes of a project. And this was a highly successful project. A summary of the primary achievements of the project is briefly highlighted here.

### 3.1.1 What We Achieved

In one year, a small team was able to:

- develop orbiter deorbit GN&C flight software
- install and enhance the simulation environment
- develop a single-string, avionics flight computer prototype using commercial parts and standards
- develop an API (Application Programming Interface)
- inspect the software
- integrate and execute the hardware, simulation and flight software with a pilot-in-the-loop, in real-time
  - OMS/TVC actuators in the loop
- perform unit, module and system level testing
- document many aspects of the project, including test plans and results
- put fundamental software infrastructure in place for multiple projects:
  - Configuration Management
  - Discrepancy Tracking System
  - Software Testing Tools
  - Software Quality Assessment Tools
  - Graphical display development tools
  - Metrics program
  - Web-based data and documentation system

A project summary report is available with additional details.

### 3.1.2 Building a Core Team

Perhaps most important for the long term, during the course of this project a capable core team for rapid development of GN&C systems evolved. This included finding talented team members; training in and gaining experience with the tools, processes

and methodology; and integrating into a cohesive team.

After working together in the RDL for a year, team members have acquired expertise in methods, tools and domain. A helpful and cooperative atmosphere has encouraged and enabled cross training. A willingness on the part of team members to address any and all project issues has proven invaluable on many occasions, as the team works through complex issues and considers all implications before choosing and acting on an alternative. Occasionally, the strong desire to obtain team consensus has resulted in long, bordering on excessive, discussions of certain issues, as should be expected among talented, highly trained professionals who assume a strong personal stake in the success of the project.

We have shown that the RDL can support this type of training and team building exercise. In addition, this team can be a long term asset to the division and to the agency.

### 3.1.3 Establishing a Benchmark

A formal effort to collect and evaluate project metrics in the RDL was launched with this project. We now have one project that has been observed and measured from start to finish. Data collected and reports generated will serve as a benchmark for later projects.

### 3.1.4 Establishing a Credible Presence

Several presentations and demonstrations of the integrated system have served to inform the JSC community about the RDL and its capability, as well as the potential of the Rapid Development methodology.

In the course of this project, the RDL and the project team made contacts with and worked with a number of organizations outside of EG, including the RAPIDS lab, SR&QA, ER, and Lockheed/Loral. These contacts will foster valuable relationships for future cooperation.

By involving other organizations and widely demonstrating the capabilities and accomplishments of the RDL, we were able to increase center awareness of both facility and staff capabilities. Further, opportunities for future cooperation and collaboration have been identified, which should leverage the return on future projects.

## 3.2 Surprises, Difficulties, and Challenges

### 3.2.1 FSSR Deficiencies

When this project began, we assumed that all detailed requirements and algorithm design were contained within the GN&C Functional Subsystem Software Requirements (FSSR) documents. Based on this assumption, we were able to justify spending minimal time on these efforts. Perhaps more significantly, developers assumed that functions could be coded with limited understanding of the overall system, by just coding directly to the specifications in the FSSRs. The team included a minimal contingent of domain experts familiar with the requirements. This deficiency was not thought to be a problem, since the requirements were believed to be extremely well documented in detail.

We discovered that, in many cases, the FSSRs do not completely or accurately represent the current flight software. Therefore, they cannot be used as the sole source for requirements information.

Also, the current FSSRs are a combination of requirement, design, and implementation details (i.e., some things in them may not actually be requirements for a new implementation). There is typically no differentiation among these types of details. In many cases, considerations that drove implementation details for the existing software may not exist in the current development and implementation environment.

As a result, modules coded precisely to the FSSRs were not always complete, correct, or efficient. Some rewrites were required. Some inefficiencies probably still exist. Some problems were not discovered until late in development.

Future similar efforts must include plans to evaluate the FSSRs for completeness, accuracy and current relevance. Sufficient domain expertise must be included in the team to allow for this, as well as time for both the analysis and to correct any observed deficiencies. We believe that this investment up front should result in better implementation and should reduce both error rates and revisions.

### 3.2.2 Compatibility Issues

One of the goals of this project and the RDL in general is to take advantage of recent industry standardization efforts to improve productivity and cut costs. Using such commonly available products as standard processors, languages, communication buses and protocols has several advantages. Some of these include:

- It is generally cheaper to purchase commercially available products than to develop a custom product in house
- Expertise with commercial products is easier to find and hire.
- Documentation and support are more readily available.
- It is usually easier to upgrade to follow advances in technology, since new

products and new versions of existing products often offer compatibility with existing standards and previous releases.

In the RDL, we also have the goal of cross-platform compatibility and easy reconfiguration to support different test environments. For example, code developed on generic workstations should port with minimal changes to flight computers, real-time test platforms, and hardware-in-the-loop configurations. Networking should allow access to on-line documentation, problem reporting system, configuration management system, test cases and tools, and metrics collection and reporting tools, from all platforms in the lab. C or Ada code should move transparently among platforms and, once compiled and linked, behave the same in all of our test and development configurations.

The reality is that, despite evolving and widely accepted industry standards, cross-platform incompatibilities still exist. Progress is evident, as the number and degree of problems encountered is now significantly decreased over just a few years ago. But each new piece of equipment, software test or development tool, and piece of hardware in the loop, must still be carefully installed and evaluated for consistency and compatibility with the remainder of the environment.

We are also concerned about the permutations of difficulty as we anticipate following upgrades, to hardware, systems software, and development tools, throughout the lab. There are implications on at least test results, configuration management, and software certification issues when, for example, an operating system upgrade takes place.

None of this should be taken as reason to abandon the approach. Rather, it is reason to continue support for industry standardization efforts. The state-of-the-practice continues to evolve rapidly, resulting in the potential for dramatic improvements in quality and productivity. We need to continue to position ourselves to take advantage of the leading edge. It would be a mistake, however, to minimize or neglect to plan for the impact of unanticipated incompatibilities, and these impacts should be addressed in project plans and schedules.

### 3.2.3 Data Flow Design

Consistent with our methodology, the first software release was primarily designed to establish the system architecture. In doing this, the team tried to specify, in the MatrixX block implementation, all of the inputs and outputs to each module in the overall architecture. In retrospect, this was not the best approach.

We thought it would simplify things to have all the interfaces defined and connected before detailed implementation started. But, in the absence of implementation details, our original analysis of what data elements were required was not complete or accurate. This approach sometimes resulted in unneeded connections, which caused confusion and impacted performance as well as system size. When required

interfaces were not included initially, the approach made it more difficult to add the pins as requirements for them were discovered. (Although this may have been an artifact of the particular development tools used, rather than the development approach.) And, specifying them individually made some of the block diagrams difficult to read and understand.

In retrospect, the team thought that the initial architecture should specify data element groups, defined by functionality and passed among blocks as a group. As the details of design and implementation evolved, adding data items to the predefined groups could yield a better overall design, and be easier to implement.

With this approach, the team still needs to carefully consider the number of data groups and what elements belong in each group, in order to have efficient data flow. It is also possible that this issue might be less important in a different development environment.

## 3.3 Quality Lessons

As the project evolved, the team concluded that we did not emphasize quality issues sufficiently in early phases of the project. These issues include, for example, unit, module and systems level tests. They also include formal inspections and the use of advanced quality assessment tools.

### 3.3.1 Inspections

This project was the first to use formal inspections as a quality enhancement tool in the RDL. Most project team members were inexperienced with inspection techniques when the project began, and several were frankly skeptical of their usefulness initially. After completing a round of inspections, most team members were solidly converted to inspection advocates.

The major problem noted about our inspection process was that it was started later than it should have been. For this project, virtually all of the code was inspected in one short time period. While individual inspections were limited to relatively small blocks of code, it became common to hold several inspections each week. Preparation time was thus limited. Also, developers expressed a desire to have more opportunity to use the results of one inspection to improve other modules before they were inspected.

The team consensus was that earlier inspections of some modules would have clarified coding standards as well as implementation issues for modules still to be implemented.

Since our methodology establishes low-fidelity end-to-end functionality early in development, and then evolves the modules to full detail over several releases, it was agreed that it is probably not reasonable to inspect all modules for all releases. We agreed on a rule of thumb that a module would be inspected when it had reached a maturity level that included substantially all of its expected functionality, or earlier if the developer of that module requested it. And that modules be reinspected if substantially modified. For now, we rely on the best engineering judgement of the developers to make these qualitative decisions on what constitutes “substantial”.

For modules developed with the MatrixX SystemBuild tool, inspections were performed on the block diagrams rather than the source code. The decision to do this was based on the premise that, if changes were made they would be to the block diagrams. (That is, we do not want to make modifications directly to code that has been autocoded, since that would remove the possibility of ever using the block diagrams and autocoder for that module in the future.) It was generally felt that this approach worked quite well.

### 3.3.2 Formal Testing

Testing strategies were not easy to define given the cyclic nature of the development. Traditional definitions and planning approaches were ineffective. This led to a typical problem of testing taking longer than anticipated.

Simply stated, we need to do more testing of early releases. Since they were not “complete” or “final”, and since there was so much development to be done, we did not emphasize formal testing for early releases of the software. Rather, we depended on individual developers to decide when the modules had been sufficiently tested for that release. Since the team was new, requirements were still evolving, and inspections were also delayed until later releases, in retrospect the team agreed that more, and more formal, testing of earlier releases should have been performed. We anticipate that final system testing will be easier and more quickly successful when this is done.

There is still a careful balance to be struck between complete testing, superfluous testing, and redundant testing, in our evolutionary spiral development model. For example, it is probably a waste of time to completely test detailed functionality of a low fidelity early release. There may not even be detailed specifications for low fidelity interim models, since they are planned to evolve to the final system. Also, some modules may change little from one cycle to the next. For this case, full scale testing may be redundant. Still, each release forms the basis for the next, so formal testing is required to assure that the basis is complete and accurate. Significant engineering judgement is required in deciding testing requirements.

For future projects, test requirements and strategy should be included with each cycle plan.

### 3.3.3 Advanced Quality Assessment Tools

Over the course of this project, the team became familiar with several advanced quality assessment tools that are either currently or soon to be available COTS products. To the extent possible, given a heavy primary work-load and tight schedule constraints, team members attempted to evaluate and use the products. Our focus was to investigate whether and how such products could be useful to developers during software implementation, and to make use of them as possible to the benefit of our project.

Some observations about these tools include:

- Overall, we learned that quality assessment tools are rapidly becoming more automated and easier to use.
- Some of the things that these tools measure, for example some complexity measurements, are still difficult to interpret for those who are not quality assurance professionals. For example, how does a developer know whether a particular complexity value is good or acceptable, what circumstances indicate a

need to modify the software, and what actions should be taken to improve it?

- Some of the available tools are prohibitively expensive and are not available across all RDL environments.
- This is an area that is coming of age, and new tools can be anticipated in the near term.

In the spirit of Rapid Development, we are looking for ways to use these tools that would solve problems early in the development cycle, when they are traditionally easier and cheaper to fix. Ideally, we want tools that capture the expertise of SR&QA professionals in a way that developers can use to enable early and accurate assessment of software quality and maturity. With this in mind, we need tools that are easy to use and interpret. We want to put these tools in the hands of developers, to augment their implementation tools and skills, and thus deliver a higher quality product for testing and inspection.

One such tool that proved particularly useful evaluated test coverage. After running test cases, this tool (X-Atac, developed by BellCore) reports the percent of code executed during the test suite. It also has the capability to report which branches were executed, and to recommend areas of the code to be exercised by added tests in order to maximize added coverage. Using this tool, developers were able to quickly evaluate and improve this aspect of test suites. Because the measurements and recommendations are easily understood, no special quality analysis training was required in order to use the tool.

The RDL should continue to look for easy to use advanced quality analysis tools to add to the developers' capabilities.

## 3.4 Project, Process and Infrastructure Issues

It was understood from the beginning of the project that formal processes, procedures and laboratory infrastructure would be evolving along with the project. Some needs, such as the use of a well defined configuration management system and policy, were anticipated in advance; we needed to better understand them in the context of Rapid Development, define the procedures and implement them. Others, such as the size of the effort required to build an effective Rapid Development core team, we learned along the way. This section highlights some important lessons of both types.

### 3.4.1 Project Planning Deficiencies

Like all too many projects, there was insufficient formal planning and schedule analysis, both as the project began and as it evolved. Some of the issues that drove this are described in the following sections.

#### 3.4.1.1 Code Now

Developers were pressured to start coding too soon. This is a perennial software development problem, not unique to any aspect of this project. For some reason, it remains difficult to convince managers that substantial progress is being made until lines of code can be counted. Rapid Development, and especially advanced development tools, may add some to this difficulty, since they tend to create an expectation that, since the team can design the system with the development tools, prototype implementation should be the first step.

It is true that, in a Rapid Development paradigm, implementation of early cycles will typically begin sooner in the life-cycle than could be expected with traditional methods. But it does not happen instantaneously.

For good project management and control, it is still critical to take the time up front to develop a work breakdown structure and project schedule. These will likely change significantly over the course of a project. Schedules must include time to re-evaluate project plans and schedules, at least at each cycle transition. We will lose some of the benefits of Rapid Development if we neglect to position ourselves to be able to provide good project oversight.

#### 3.4.1.2 Demonstrate Now

This was an exciting project. Both the team and management wanted to show it off. People outside the team and the division were, and remain, interested in our progress and results. These are all good things, but they sometimes got out of hand.

Too often, the need to provide specific demonstrations of capability drove development priorities. These interruptions effected, for example, testing of early

cycles. The team probably over-developed for early cycles, including too much new functionality in each cycle, in an effort to improve the demonstrations. (We all know that we may not get another chance with a particular audience, so want to show them all the best stuff!) This focus left little time for testing before the demonstration, so sometimes we were forced to test to the specific desired demonstration functionality rather than overall functions. This approach was not an efficient use of development or testing effort.

We must do a better job of driving our priorities by the needs of the project. The demonstrations need to fall out of, or follow, a project plan, not drive it.

### 3.4.1.3 Solve Everything

We began with a very broad scope. We wanted to test tools, evolve infrastructure, mature the methodology, compare languages, compare hardware, investigate new techniques, build a Rapid Development core team, and develop flight software. We wanted to do it all “better, faster, cheaper”. And we added expectations and goals as the project evolved and new understanding suggested new ideas.

This was not necessarily bad for this initial demonstration project. But it is worth noting that future projects would benefit from a less inclusive, more focused, and less volatile scope.

### 3.4.1.4 Do It Immediately

Throughout this project, schedules were demanding and overly optimistic. Sometimes this led to schedule slips. Often it led to long hours for the development team. While the team is willing and dedicated, this sort of schedule is not sustainable. Some of the scheduling difficulties were certainly driven by management pressure to see results quickly. But more often the schedules were a result of optimism and desire on the part of the team, coupled with immaturity of rapid development cost and scheduling techniques overall. The degree to which we had to understand and interpret the FSSRs, rather that just code to them as originally planned, was also a significant schedule challenge.

The effects of preparing for demonstrations had significant impact as well. When these are not in sync with the delivery schedule, the team is effectively required to produce an unplanned release. This delivery is a costly disruption, and is seldom anticipated in the schedules. If demonstrations of “latest and greatest” capability continue to be an important part of these projects, the schedules must build in time to prepare for them, and management must understand the costs to the project. A better approach, from the project point of view, is to freeze demonstration versions between releases, but it is recognized that this is not always possible due to considerations outside the project.

### 3.4.2 Infrastructure Considerations

The RDL is a relatively new facility, and our methodology is still evolving. When the project started, some important support infrastructure was lacking. We knew this from the outset, and in fact one of the project goals was to better understand and implement some of the required infrastructure. Infrastructure areas that proved to be the most significant include:

- insufficient configuration management
- insufficient test and problem tracking capability
- insufficient licenses for important development and testing tools
- some team members not familiar with real-time GN&C systems
- some team members not familiar with Rapid Development

Over the course of the project, significant progress was made in all of these areas.

Configuration management, and test and problem tracking, are now handled through a combination of COTS and in-house developed tools which are on-line, automated, and available to all team members. We still need to fine tune our procedures, but overall we see significant improvement. The facts that these tools are easy to keep up to date, and easy for all team members to access, works well with the overall Rapid Development approach.

It is common today for COTS software products to be “license limited”. That is, at any given time, the total number of users of a particular product (including each developer, tester, project manager, etc.) cannot exceed the number of copies, or licenses, that have been purchased for that product. The cost of purchasing multiple licenses is usually significant. But the potential loss of productivity, if insufficient licenses are available when required, can also be quite costly.

As part of improving the infrastructure, the RDL continues to improve license availability. When planning for Rapid Development projects, it will continue to be important to assess the need for COTS licenses and the availability, cost, and lead time to acquire sufficient licenses. We learned that operational issues can have impact, as well. For example, our configuration management system will hold a released license for a minimum of 30 minutes before making it available for reassignment. During testing especially, this can cause delays.

We initially underestimated the effort required to build an effective GN&C Rapid Development team. We should not attempt to do it from scratch for every project, but should try to maintain a working core team as a lab and agency asset.

### 3.4.3 Requirements Evolution Issues

There was some considerable confusion with early cycles especially, about the goals and capabilities to be delivered. We mostly discussed those issues, without much documentation. In retrospect, some additional documentation was in order.

Future projects need to be more rigorous about writing release plans. At least the expected deliverable for the current cycle should be very well defined and bounded. We found that too often team members tried to include more capability into each cycle than was actually required for that delivery. Often this approach backfired and delayed the cycle. We anticipate that this tendency will be reduced as both the team and project management become more familiar with, and more confident in, the Rapid Development evolutionary spiral development approach.

## 3.5 Other Rapid Development Methodology Issues

### 3.5.1 Importance of Early End-to-End Architecture Cycle

The evolutionary spiral model of the Rapid Development method calls for, in the first implementation cycle, a low-fidelity, end-to-end version of the software to establish and validate the planned system architecture. At the beginning of this project, the team was divided in discussions about the need to do this.

The need for the initial iteration, defining the architecture, was arguable based on the assumption that the system design was already complete and detailed, per the FSSRs. In some ways, it seemed that we had too much fore-knowledge to really exercise Rapid Development the way it was meant to be used.

But as the project evolved, we discovered deficiencies in the FSSRs as detailed design documents. Also, as developer expertise with the development tools and with the real-time GN&C application increased, there also came increased understanding of subtle issues that, had they been taken into account in the initial cycle, could have eased later implementation issues. (The data flow issue discussed previously is one.)

In the end, there were a lot of converts. It remains an important tenant of our Rapid Development methodology that significant effort to understand and stabilize a solid system architecture should be invested before significant detailed implementation efforts begin.

### 3.5.2 Utility Libraries

We found, as the project evolved, that both design and implementation could be improved through the use of some standard utilities. These utilities were eventually collected into a project library area. In retrospect, this issue should probably have been addressed early in the project for maximum effectiveness. Future revisions to the Rapid Development guidelines should recommend that the need for and implementation of a library of standard utility functions be addressed after the initial architecture implementation cycle is completed, and revisited as needed throughout a project. Not every project will require this, but for those that do, setting these libraries in place early reduces redundant coding and helps standardize common functions.

### 3.5.3 Configuration Management Issues

Historically, configuration management (CM) has been used as a gateway to make sure that only approved software passes into the system. Typically, changes made to software under CM must be correlated to change requests (CRs) or discrepancy reports (DRs), and must receive management or board level approval before being accepted.

With this sort of experience, we were inclined to delay putting the software under configuration management until it was nearly completed and had begun to stabilize. We did not want to interfere with the momentum of rapid development in the early cycles.

But this had problems of its own. Changes came fast, and various developers needed to coordinate to make sure that everybody was using and testing with the current version. And to make sure that changes made by more than one developer did not conflict or cancel each other.

We settled on a configuration management approach that is more open to the developers, somewhat less rigorous in terms of control, but quite powerful in terms of tracking changes, integrating work of multiple developers, and coordinating multiple releases. A COTS tool, ClearCase, is used to support CM for the project. It can be strongly coupled with another COTS tool, DDTS, which we are now using to report and track CRs and DRs against the software.

By making it relatively easy to check out, modify, and check in software, the use of this CM process places relatively little demand on developer productivity, while providing excellent protection of the evolving software. The major complaint expressed by developers relates to the issue of license availability rather than any problem with the tool or process itself. As the software matures, and passes through more testing and approval gateways, control over software modifications can be increased through options available in the software.

The lesson learned is that configuration management in the early stages of Rapid Development can be used effectively to the benefit of projects. It is important to strike a balance between control and protection of project assets, while maintaining enough flexibility to support a highly dynamic work environment. This type of intermediate and flexible control can be achieved with currently available support software.

### 3.5.4 Performance Testing Issues

Our early definition of the Rapid Development methodology approached performance testing and issues as part of the final system tune-up, to be addressed when implementation of functionality was nearly completed.

In this project, we discovered that approach may have resulted in some early design mistakes. Real time considerations are significant and need to be addressed early and often in the development. Performance testing should be part of every release. When addressing observed performance problems, however, it is not always appropriate to insist on meeting performance benchmarks for every release. Sometimes it may be sufficient to explain the source of any observed problems, especially in cases where a later release will specifically address improvements in the fidelity and performance of the problem area.

### 3.5.5 Auto-Coding Issues

Rapid Development does not depend on the use of auto-coding technology. In practice, however, we are finding that advanced development tools with auto-coding capability are tremendously useful in a Rapid Development environment.

Many questions arise when using auto-coders for flight software. To what degree is it necessary to examine, inspect, or otherwise verify the resulting code? When tests have been performed using the block diagram implementation, what criteria determine whether to repeat tests with the auto-coded software? How can we certify or validate an autocoder? What approach do we take to following updates to autocoders and to recertification issues when the autocoder has a new release?

As this project concludes, we still do not have definitive answers to these questions. For now, we have settled on some working rules of thumb. It has seldom proved useful to invest much time in visually examining autocode, since it is not very readable. We perform formal testing on the coded version, even if the tests have previously been performed on the block diagram version. We have observed some minor discrepancies between system behavior before and after autocoding.

We observe that the state of this technology is rapidly advancing. As it does, we anticipate increased confidence it the accuracy and dependability of autocode. These useful tools should one day be like today's compilers, necessary and dependable tools of our trade. In the meantime, it seems prudent to err on the side of increased caution and testing.

### 3.5.6 Document Coordination Issues

Throughout the project, we suffered from a variety of coordination issues with respect to electronic and paper documentation. These issues related to composition, distribution, availability, and status. The root of the problems was generally platform and format compatibility. While it sounds simple enough to decree that all documents shall use a particular format and be distributed electronically, the reality is that we work in a multiple contractor environment and incompatibilities continue to exist. These include platform (e.g., Mac, PC, Unix), tool (e.g., Word, FrameMaker), and even version (Word 4 vs. Word 6, for example) difficulties. Then, as we try to distribute the documents, there are sometimes e-mail system compatibility issues.

A major breakthrough in this area has been the use of project web pages to file all project documents. All project documents and files have been stored as either plain ascii files (these are primarily test results) or as PDF (Portable Document Format) files. These formats are easily read by all the major platforms, using standard web browsers (such as NetScape or Explorer) with an Acrobat reader plug-in.

Aside from cross-platform readability, the web pages also serve to ameliorate

problems of document version control. Any team member can access the most recent document versions at all times. By focusing on electronic documentation, we have minimized occurrences of using obsolete document versions.

The success of this effort depended on the efforts of a team member who was responsible for maintaining the web pages. Once the team begins to depend on them, it is critical that these pages be accurate, organized and up to date.

We still struggle with compatibility issues in some areas. For example, we cannot perform revisions to the documents in PDF format, so must revert to the originating document file. This is also true when, for example, a figure, table, or text from one document is used in another document. Thus we must maintain at least two versions of documents (PDF and original). And any compatibility issues persist with the original documents.

The good news appears to be that cross-compatibility issues are being addressed in the document preparation products, so we are beginning to see improvement. It is still advisable to address these issues early, to try to minimize the problem.

### 3.5.7 Systems Expertise

Our methodology and tools put a great deal of development capability into the hands of domain experts. This leverages effort spent by these domain experts to develop and design system and control algorithms. Since we never throw the design “over the wall” (to developers), we eliminate the step of algorithm re-coding, which eliminates both redundancy of effort and a possible source of errors.

It would be a mistake, however, to assume that these domain experts can replace developer and systems expertise completely. There is more to good software system design and implementation than the core algorithms. Systems expertise, especially in architecture, interface, and data flow issues, is needed to produce testable, maintainable and error free systems.

## 3.6 Metrics Insights

Metrics are collected in order to measure, track, manage and plan projects. The successful use of metrics depends on knowing what to measure, taking accurate measurements, and interpreting those measurements. Much of this is predicated on having historical data for comparison. For Rapid Development, there is very little available historical data, including the large commercially available project metrics data bases. We, along with others in the software systems field, are developing appropriate methods as we go, while trying to take what we can from metrics programs that have been developed for traditional methodologies.

Details of metrics collected and data results are presented in the Project Summary and in a separate Metrics report. Here, we will emphasize some of the general lessons to be learned from our efforts.

Perhaps most important, we were able to collect a significant amount of meaningful project data while using a Rapid Development approach. This both shows that it can be done, and provides an initial baseline for building, as a very long term goal, project data bases and estimation and management guidelines.

This is a preliminary effort. There is not yet sufficient data or experience to draw significant conclusions about the best way to use metrics in Rapid Development. We are still looking primarily at output metrics, which monitor and record project progress, as is appropriate for an early effort. As we learn more, we hope to be able to concentrate more on outcome metrics, which can be used to position a project for success more pro-actively.

Among the specific lessons learned from this effort are:

- The team decided that specific development resource utilization metrics were not only difficult to collect but probably irrelevant in our development mode. CPU time, for example, is not a typically scarce or limiting resource. And those resources that were limiting were obvious: insufficient licenses for some development tools, and only one full-blown real-time test platform available. When we get more sophisticated at this, we may want to revisit this type of metric, to help us determine, for example, how many additional licenses are optimal.
- A more relevant and critical resource measurement for our needs is the performance of the developed software. This is due to the real-time demands of GN&C systems.
- Code size measurements, whether as SLOC or MatrixX blocks, are not predictive of project progress for Rapid Development, since the code tends to expand rapidly in early iterations and then stabilize during cycles which refine functionality. Code size may still be a good predictor of cost or schedules, so we choose to continue to measure it to populate our historical data base.
- Function Point measurements should be further investigated, but require significant training and understanding for successful use.
- Automated, semi-automated, or on-line tools can be developed to both improve

the accuracy and currency of data and minimize the impact, on team members, of capturing the data. When such tools are used, they need to be captured under configuration management, to assure consistent measurements throughout the project life-cycle.